UNIVERSIDAD REY JUAN CARLOS

TRABAJO FIN DE GRADO

Aplicación para la compartición segura de ficheros

Autor:
Sergio Merino Hernández

Tutor:
Dr. Gorka Guardiola Múzquiz

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN
GRADO EN INGENIERÍA EN TELEMÁTICA

7 de junio de 2018

Resumen

La comunicación digital ha supuesto un increíble avance en la transmisión de información. Sin embargo, esto también ha desencadenado nuevas formas de robarla y falsificarla. Para solucionarlo existen multitud de algoritmos de seguridad que permiten cifrarla y firmarla, con el fin de mantener su confidencialidad, su integridad y su autenticidad.

Gracias a estos algoritmos surgen múltiples aplicaciones que permiten la comunicación preservando estas tres propiedades. Estas aplicaciones existen para la inmensa mayoría de las plataformas actuales y hacen uso de la combinación de diversos algoritmos.

En este contexto, este Trabajo de Fin de Grado se enfoca en el desarrollo de una aplicación que permita llevar a cabo comunicaciones seguras entre usuarios. Esta aplicación se va a desarrollar para el sistema operativo Android y va a hacer uso de varios algoritmos de cifrado estándar como RSA o AES.

Índice general

Índice de figuras

Índice de cuadros

Lista de Abreviaciones

AES	**A**dvanced **E**ncryption **S**tandard
API	**A**pplication **P**rogramming **I**nterface
ART	**A**ndroid **R**unt**i**me
BC	**B**ouncy **C**astle
CBC	**C**ipher **B**lock **C**haining
CFB	**C**ipher **F**eed**b**ack
ECB	**E**lectronic **C**ode**b**ook
EOF	**E**nd-**O**f-**F**ile
HMAC	**H**ash-based **M**essage **A**uthentication **C**ode
HTTP	**H**yper**t**ext **T**ransfer **P**rotocol
IV	**I**nitialization **V**ector
JCA	**J**ava **C**ryptography **A**rchitecture
JCE	**J**ava **C**ryptography **E**xtension
JIT	**J**ust **I**n **T**ime
JVM	**J**ava **V**irtual **M**achine
MAC	**M**essage **A**uthentication **C**ode
MCD	**M**áximo **C**omún **D**ivisor
MD	**M**essage **D**igest
mcm	**m**ínimo **c**omún **m**últiplo
OFB	**O**utput **F**eed**b**ack
PSS	**P**robabilistic **S**ignature **S**cheme
QR	**Q**uick **R**esponse
RFC	**R**equest **f**or **C**omments
RSA	**R**ivest - **S**hamir - **A**dleman
SCP	**S**ecure **C**opy **P**rotocol
SHA	**S**ecure **H**ash **A**lgorithm
SSA	**S**ignature **S**cheme with **A**ppendix

1 Introducción

1.1. Motivación

Las comunicaciones seguras nacen del deseo de protegernos: de proteger con quién nos comunicamos y qué comunicamos.

De este deseo surgen multitud de algoritmos de seguridad que hoy en día usamos sin darnos cuenta. Desde una simple consulta web hasta la felicitación de Año Nuevo, nuestras comunicaciones pasan por diversas operaciones para preservar su seguridad.

Esta seguridad viene generalmente proporcionada por la confianza que depositamos en ciertas organizaciones, entidades que crean una red de confianza sobre la que se sustenta todo este sistema.[1] Pero, ¿qué sucede si de quién nos queremos proteger es de ellos? ¿Por qué tenemos que confiar en que una entidad gubernamental sea la que preserve la seguridad de nuestras comunicaciones? ¿Por qué no podemos tener nuestra propia red de confianza?

Con esta idea comienza el desarrollo de una herramienta que nos permita mantener comunicaciones seguras sin depender de una Autoridad Certificadora para ello.

1.2. Estructura de la memoria

La estructura que se va a seguir en este proyecto es la siguiente:

- En el Capítulo 2 se presentan los objetivos que se persiguen con este proyecto y un modelo de amenaza para el mismo.
- En el Capítulo 3 se explican varios conceptos y tecnologías que ya existen y que se han usado para llevar a cabo el proyecto.

[1] Nos referimos a las Autoridades Certificadoras.

- En el Capítulo 4 se detallan los diferentes prototipos y la arquitectura de seguridad de la aplicación, así como el tiempo dedicado a cada prototipo.
- En el Capítulo 5 se exponen los resultados obtenidos con ejemplos y los problemas encontrados durante el desarrollo del proyecto.
- En el Capítulo 6 se finaliza la memoria haciendo una reflexión sobre los resultados y las posibles líneas de desarrollo futuras.

2 Objetivos

2.1. Objetivo general

El objetivo principal es el desarrollo de una aplicación para el sistema operativo Android, que permita a cualquier usuario realizar una comunicación de datos de manera que se preserve la confidencialidad, la integridad y la autenticación de la información transmitida.

2.2. Objetivos específicos

Para abordar el objetivo principal del proyecto, este se ha dividido en unos objetivos más específicos:

- Proporcionar un mecanismo que permita la transmisión de información entre distintos dispositivos.
- Desarrollar un esquema que permita cifrar y descifrar la información que se quiere transmitir.
- Diseñar una forma de comprobar la integridad y la autenticación de la información.

2.3. Modelo de amenaza

En criptografía, un modelo de amenaza (*threat model*) especifica que tipo de amenazas potenciales a un sistema puede realizar un individuo (o grupo) con unos determinados recursos. Con el objetivo de proteger el sistema de estas amenazas, y únicamente estas, se diseña un determinado esquema de seguridad para poder contrarrestarlas.

Nuestro modelo de amenaza será un atacante con conocimientos de criptoanálisis y con una capacidad de cómputo limitada, como la que puede tener un usuario normal en casa o en una oficina.

Los puntos más vulnerables de nuestro modelo son el *hardware* y el *software* sobre el que ejecuta el sistema, por lo que dejamos fuera a aquellas compañías que han trabajado en su desarrollo, como Intel, Google, Qualcomm, etc.

Debido a los recursos casi ilimitados que poseen también dejamos fuera de nuestro modelo a cualquiera de los gobiernos de las grandes potencias del mundo, como China, Corea, Rusia o Estados Unidos.

Un atacante de nuestro modelo de amenaza podría realizar los siguientes tipos de ataque:

- Ataques sobre el texto cifrado – En este tipo de ataques el atacante dispone únicamente del texto cifrado y no tiene acceso al texto plano. Ejemplos de este modelo son los ataques por fuerza bruta, en los que se prueba cada una de las posibles combinaciones para una clave hasta dar con la correcta.
- Ataque de texto plano conocido – En este tipo de ataques se presupone que el atacante tiene acceso a un número limitado de textos planos y sus correspondientes textos cifrados. El objetivo de este tipo de ataques es el de obtener la clave de cifrado a partir de estos pares, con el fin de poder descifrar futuras comunicaciones.
- Ataque de texto plano selectivo – En este tipo de ataques el atacante puede seleccionar un número indeterminado de textos planos y obtener sus equivalentes cifrados. Con esto, un atacante puede probar distintas combinaciones y encontrar patrones sobre el texto plano para explotar alguna vulnerabilidad.
- Ataque de texto cifrado selectivo – Parecido al anterior, el atacante puede obtener un número indeterminado de textos planos a partir de sus equivalentes cifrados.
- Ataques sobre la clave – En este tipo de ataques el atacante dispone de alguna información acerca de la clave de cifrado.
- Ataque de intermediario (*Man in the middle*) – En este tipo de ataque el atacante retransmite o modifica la información que un usuario envía a otro, haciendo creer a ambas partes que mantienen una comunicación directa.

- Ataques de canal lateral (*Side channel*) – Este tipo de ataques explotan vulnerabilidades físicas del sistema informático (en lugar de basarse en debilidades del algoritmo utilizado), tales como fugas electromagnéticas o análisis de diferencias del consumo energético.

[28]

3 Estado del Arte

3.1. Java

Java es un lenguaje de programación de propósito general, orientado a objetos y concurrente. Originalmente fue desarrollado por James Gosling, Bill Joy y Guy Steele para Sun Microsystems en 1996 y fue adquirido por Oracle en 2010.

Fue diseñado para que los desarrolladores escribiesen una única vez su programa y pudieran ejecutarlo en cualquier máquina sin necesitar recompilarlo. Esto es posible debido a que las aplicaciones Java son compiladas a *bytecode* que luego es ejecutado en una *Java Virtual Machine* (JVM), sin importar la arquitectura de la máquina. *[9]*

FIGURA 3.1: Logo de Java *[21]*

3.2. Android

Android es un sistema operativo desarrollado por Google, basado en el *kernel* de Linux. Está diseñado principalmente para dispositivos táctiles, como *smartphones* y *tablets*.

Las aplicaciones de Android están escritas en Java. Hasta la versión 4.4 de Android, se utilizaba Dalvik como máquina virtual con la compilación en tiempo de ejecución (JIT) para ejecutar *Dalvik bytecode*, que es una traducción del *Java bytecode*.

Android 4.4 introdujo el ART (*Android Runtime*) como un nuevo entorno de ejecución, que compila el *Java bytecode* durante la instalación de una aplicación. Desde Android 5.0 se convirtió en la única opción en tiempo de ejecución. *[23]*

FIGURA 3.2: Logo de Android *[20]*

3.2.1. *Keystore*

El *Keystore* de Android es un sistema que permite almacenar claves criptográficas en un contenedor de manera que resulte más difícil extraerlas del dispositivo.

Las claves que se encuentran en el *Keystore* están protegidas contra la extracción mediante dos medidas de seguridad:

- En primer lugar, las claves nunca ingresan al proceso de la aplicación. Cuando una aplicación utiliza alguna de estas claves para llevar a cabo una operación criptográfica, el texto sobre el que se va a realizar la operación es llevado a un proceso ajeno a la aplicación que se encarga de realizar la operación criptográfica. De esta manera, si el proceso de la aplicación se ve comprometido por un atacante, este no será capaz de extraer las claves.
- En segundo lugar, las claves se protegen mediante *hardware* seguro del dispositivo Android. Este *hardware* evita que las claves puedan ser extraídas incluso cuando el sistema operativo se ve comprometido: un atacante podría utilizar las claves ligadas a una *app* para realizar operaciones criptográficas, pero nunca podría extraerlas del dispositivo.

[6]

3.3. Bouncy Castle

Bouncy Castle (BC) es una API utilizada en criptografía. Esta API, entre otras cosas, proporciona los siguientes servicios:

- Una API criptográfica ligera para Java y C#.
- Un proveedor para *Java Cryptography Extension* (JCE)[1] y para *Java Cryptography Architecture* (JCA).

BC está mantenidos por una organización caritativa australiana, conocida como *The Legion of the Bouncy Castle*. *[3]*

3.4. Criptografía de clave simétrica

La confidencialidad en las comunicaciones es el objetivo principal que se persigue con los algoritmos de cifrado. La criptografía de clave simétrica nos permite comunicarnos con otra persona (o máquina) de manera que un tercero, aun teniendo el mensaje cifrado, no pudiera extraer nada de información de él.

Los algoritmos basados en este modelo utilizan un secreto compartido entre los dos extremos que se quieren comunicar. Una vez acordado, nadie que no posea este secreto podría descifrar ningún mensaje enviado por uno u otro extremo. (Figura 3.3)

El problema que tiene este modelo es el intercambio de las claves. Es necesario un canal seguro por el que comunicar las claves y este tipo de algoritmos no proveen ese servicio. *[5]*

Es por ello que normalmente se mezcla este tipo de criptografía con otro conocido como criptografía de clave pública, que veremos más adelante.

[1] JCE implementa cifrado, generación y protocolos de establecimiento de claves y algoritmos MAC.

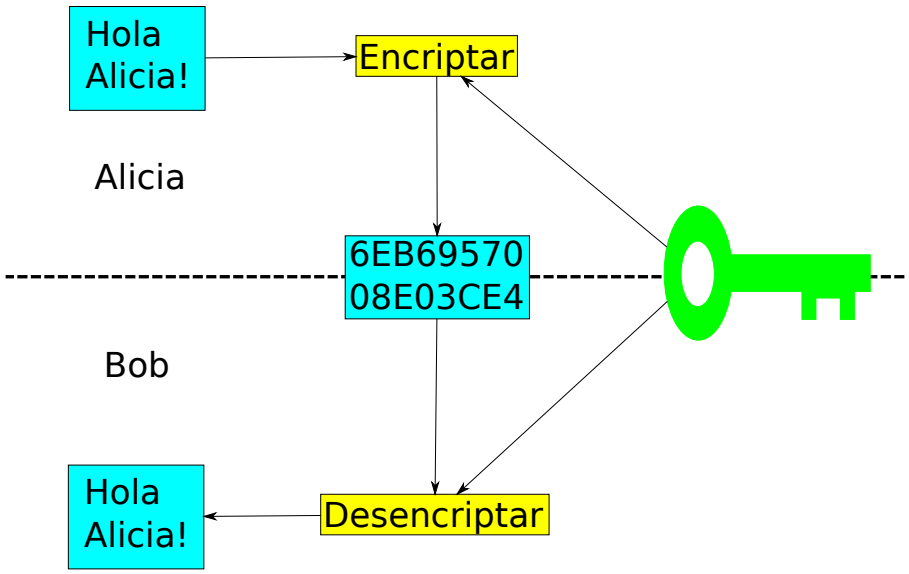

FIGURA 3.3: Esquema general de la criptografía de clave simétrica

3.5. Cifrado por bloques

Dentro de la criptografía simétrica nos encontramos dos grandes algoritmos: los de flujo y los de bloques. Nos centraremos en el segundo.

Un algoritmo basado en cifrado por bloques es aquel que, como su propio nombre indica, realiza un cifrado primero dividiendo el texto plano en bloques de un tamaño determinado[2] y luego cifrando por separado estos bloques, dando como resultado un texto cifrado.

La mayoría de los algoritmos de cifrado por bloques son iterativos, es decir, realizan la misma operación un determinado número de veces o rondas (*rounds*). Esta operación suele ser idéntica en todas las rondas, a excepción de la primera o la última, donde suele ser distinta.

Existen varios modos de operación para llevar a cabo un cifrado por bloques: ECB, CBC, OFB, CFB, etc. Cada uno de ellos opera los bloque de diferente forma: algunos utilizan el bloque cifrado anterior para generar el nuevo, otros utilizan combinaciones con estructuras del mismo tamaño que el bloque, etc. *[4]*

[2]Cuando el tamaño del fichero que queremos cifrar no es múltiplo del tamaño de bloque, el bloque final tendrá un tamaño diferente al resto. Esto se soluciona con técnicas de *padding*, lo cual se explica en el siguiente punto.

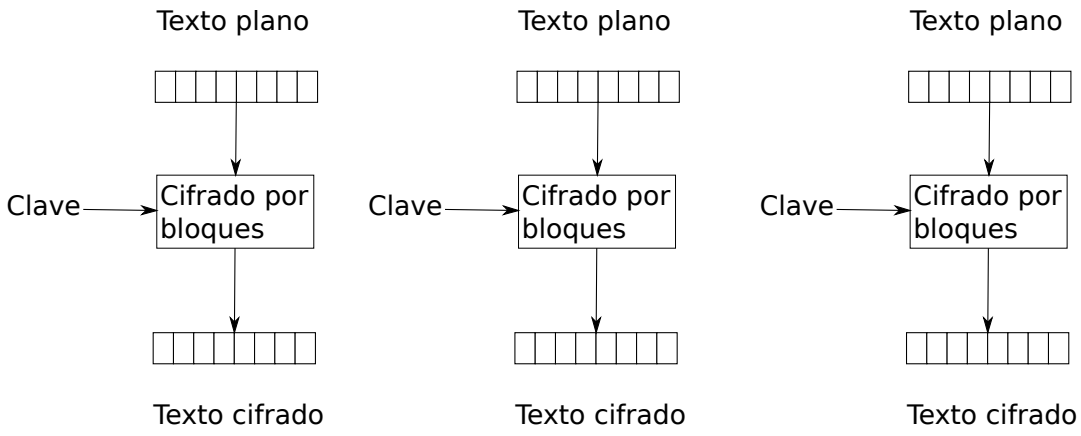

FIGURA 3.4: Cifrado usando el modo *Electronic Codebook* (ECB)

3.6. Relleno (*Padding*)

Relleno (*Padding*) es el nombre que recibe la técnica que permite en criptografía por bloques expandir el último bloque del mensaje hasta lograr un tamaño deseado.

Es muy común que, en la criptografía por bloques, los fragmentos que ciframos no tengan la longitud que queremos para nuestro sistema. Para solucionar esto existen multitud de técnicas de *padding*, desde agregar al final del bloque un *byte* con un cierto valor, hasta simplemente rellenar con ceros.

El requisito indispensable que debe cumplir cualquier técnica de *padding* es que debe permitir al destinatario diferenciar los *bytes* del mensaje original de los *bytes* de relleno. *[1]*

3.7. CBC

Cipher Block Chaining (CBC) es uno de los modos de operación para cifrado por bloques que hemos mencionado antes, y el que se ha decidido elegir para este proyecto.

En algunos modos como ECB, dada una clave determinada, cualquier *plaintext* siempre dará como resultado el mismo *ciphertext*. Si, como es nuestro caso, esta característica supone un problema, se opta por otros modos de operación como CBC, el cual soluciona esto. *[24]*

3.7.1. Modo de operación

CBC funciona combinando cada bloque de *plaintext* con el bloque de *ciphertext* justamente anterior. Obviamente, para el primer bloque no se dispone de un bloque cifrado anterior, por lo que se recurre al uso de un vector de inicialización (IV).[3]

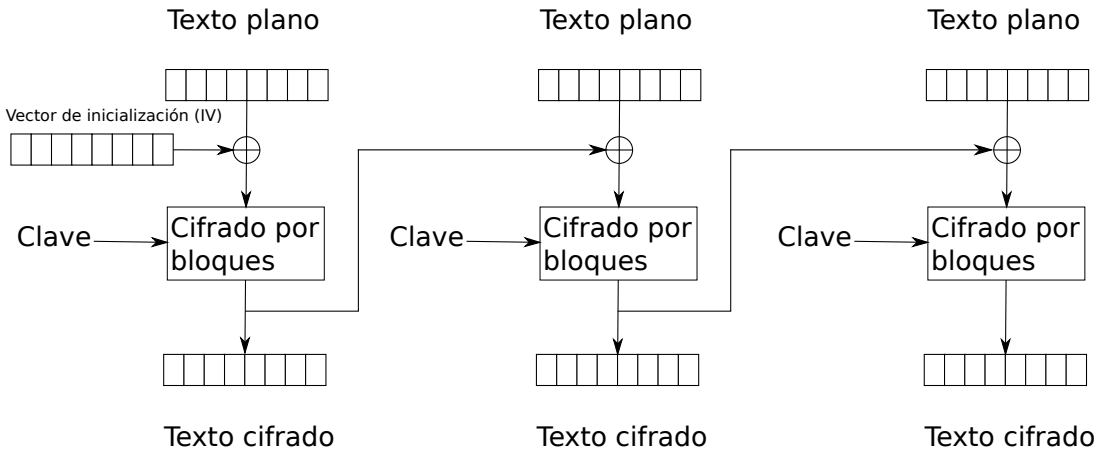

FIGURA 3.5: Cifrado usando el modo *Cipher Block Chaining* (CBC)

Como vemos en la Figura 3.5, en el cifrado con CBC, el primer bloque de texto y el IV son sometidos a una operación XOR. El resultado de esta operación se pasa por una función de cifrado[4], la cual nos dará el primer bloque cifrado. Este primer bloque es entonces pasado por un XOR junto con el segundo bloque de texto, y así sucesivamente.

A la hora de descifrar, la misma función utilizada en el cifrado es usada para obtener cada bloque de texto a partir de los bloques cifrados. Una representación de este proceso lo encontramos en la Figura 3.6.

El objetivo de un cifrado por bloques es poder paralelizar el proceso de cifrado. Lamentablemente, el cifrado con CBC debe ser secuencial, ya que para obtener cada bloque es necesario haber generado antes el anterior.

[3]Un IV es un conjunto de *bytes* aleatorios que se usan para suplir la necesidad de un bloque cifrado anterior al primer bloque. Una muy mala práctica, por no decir prohibida, es la reutilización de un IV, ya que este debe ser de uso único.

[4]La función de cifrado es la misma para todos los bloques.

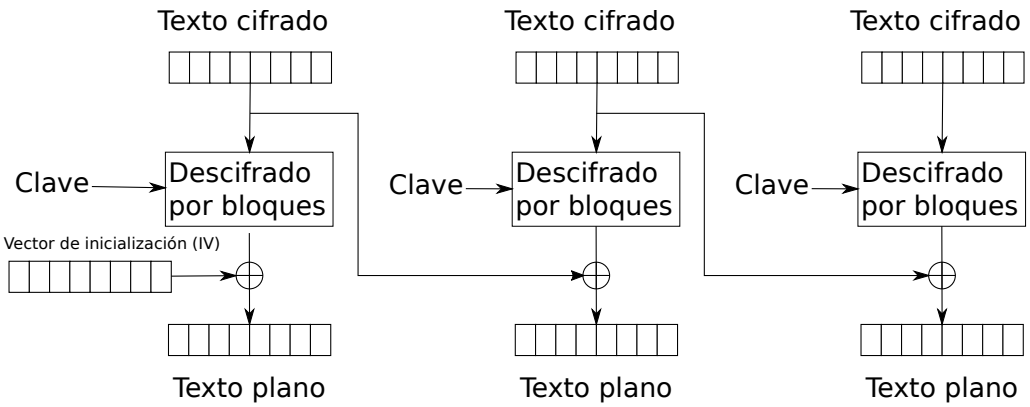

FIGURA 3.6: Descifrado usando *Cipher Block Chaining* (CBC)

Sin embargo, el proceso de descifrado con CBC sí que puede ser paralelizado, ya que las múltiples funciones de descifrado que se realizan no dependen de ningún bloque anterior. *[7]*

3.8. AES

Advanced Encryption Standard (AES), también conocido como Rijndael por sus creadores, es el resultado del proceso de búsqueda de un estándar de cifrado por parte del gobierno de los Estados Unidos.

AES es una variante de Rijndael (que a su vez es una variante de Square), del cual solo toma algunos modos. Mientras que el algoritmo original puede tomar tamaños de bloque múltiplos de 32 *bits*,[5] AES únicamente opera con tamaños de bloque igual a 128 *bits*.

Con el tamaño de las claves ocurre algo parecido, ya que AES solo soporta tamaños de 128, 192 y 256 *bits*, mientras que el algoritmo original soporta unos cuantos más. *[22]*

3.8.1. Estado (*State*)

Antes de meternos a hablar del algoritmo en sí, es necesario explicar una estructura que se usará constantemente.

[5]Con un mínimo de 128 *bits* y un máximo de 256.

El Estado (*State*) es una estructura bidimensional de *bytes* con la que se operan los *bytes* de los bloques de entrada. Está compuesto por 4 filas y 4 columnas, dando un total de 32 celdas, en las cuales se almacenan los 16 *bytes* que forman un bloque. *[14]*

3.8.2. Transformaciones

El algoritmo consta de ciertas fases, una de ellas consiste en someter el *State* a unas cuantas rondas de transformaciones. Estas transformaciones son las siguientes:

- ***SubBytes*** – Mediante una transformación no lineal, los *bytes* del *State* son reemplazados por otros usando una tabla de sustitución. (Figura 3.7)
- ***ShiftRows*** – Los *bytes* de las 3 últimas filas del *State* se desplazan de manera cíclica, cada fila con un *offset* distinto. (Figura 3.8)
- ***MixColumns*** – Mediante una transformación lineal, las columnas del *State* se mezclan para producir unas nuevas. (Figura 3.9)
- ***AddRoundKey*** – En esta transformación, una *Round Key* se añade al *State* mediante una operación XOR. La longitud de la *Round Key* debe ser igual al tamaño del *State*. (Figura 3.10)

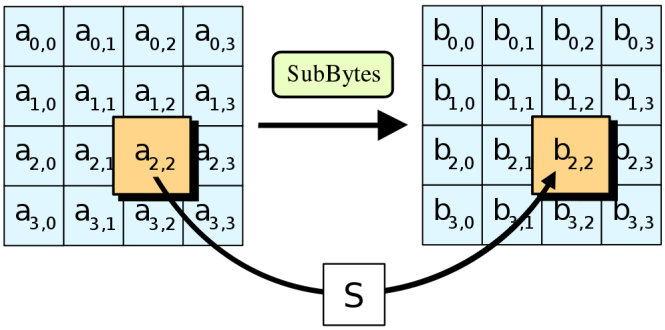

FIGURA 3.7: Operación *SubBytes* para AES *[19]*

[14]

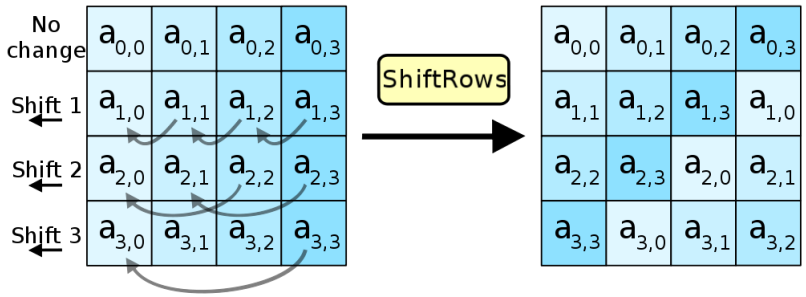

FIGURA 3.8: Operación *ShiftRows* para AES *[18]*

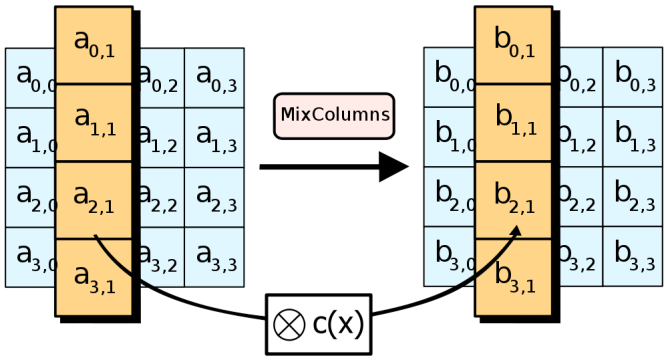

FIGURA 3.9: Operación *MixColumns* para AES *[17]*

3.8.3. Algoritmo

En AES el tamaño del bloque de entrada, del de salida y del *State* es de 128 *bits*, y el tamaño de la clave puede tomar los valores de 128, 192 ó 256 *bits*.

Para cada bloque de entrada, la primera etapa del algoritmo consiste en generar las *Round Keys* a partir de la clave, usando el esquema de claves Rijndael.[6]

Una vez conseguidas las *Round Keys*, se pasa por una ronda inicial especial en la cual solo se realiza la transformación *AddRoundKey*.

Después de esta etapa inicial, se pasa a las rondas habladas en el punto anterior. El número de rondas que lleva a cabo el algoritmo depende del tamaño de la clave, lo cual vemos representado en el Cuadro 3.1.

[6]Este esquema expande una clave en un número determinado de claves separadas.

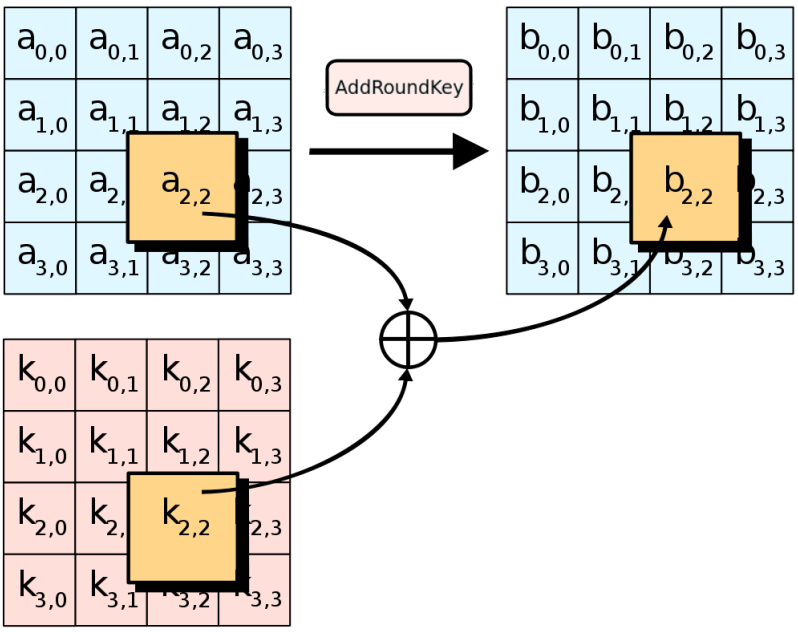

FIGURA 3.10: Operación *AddRoundKey* para AES *[16]*

CUADRO 3.1: Combinaciones para el número de rondas en AES.

***Key size* (*bits*)**	***Block size* (*bits*)**	***Rounds* (Nr)**
128	128	10
192	128	12
256	128	14

En todas estas rondas menos en la última, el orden de las transformaciones será siempre el mismo:

$$SubBytes \rightarrow ShiftRows \rightarrow MixColumns \rightarrow AddRoundKey$$

La última ronda es idéntica a las anteriores salvo por el hecho de que la transformación *MixColumns* no se realiza.

El resultado de todo esto será el *output* de nuestro bloque de entrada. *[14]*

3.9. Criptografía de clave pública

Uno de los problemas que tiene la criptografía de clave simétrica es que usamos la misma clave para el cifrado y el descifrado, por lo que se hace necesaria su ocultación. En la criptografía de clave pública, al hacer uso de dos claves, no es necesario mantener en secreto ambas. Aquella que ocultamos es llamada clave privada y la otra, pública.

La esencia de este algoritmo radica en que un mensaje cifrado con una clave pública solo puede ser descifrado con su homóloga privada, y viceversa. De esta manera podemos, tal como su propio nombre indica, difundir públicamente nuestra clave pública mientras mantenemos oculta la privada.

Si lo que queremos es confidencialidad durante la comunicación, entonces ciframos el contenido del mensaje con la clave pública del destinatario, de forma que solo él podrá descifrarlo (Figura 3.11).

Por otra parte, también nos interesa que cuando alguien reciba nuestro mensaje pueda estar seguro de que realmente es nuestro. Para lograr esto, lo que hacemos es cifrar un resumen del mensaje (*hash*) con nuestra clave privada. De esta forma, cuando alguien lo descifre con nuestra clave pública y compruebe el resumen, podrá estar seguro de que proviene de nosotros.[7] Así conseguimos preservar la integridad y la autenticación del mensaje, dos parámetros muy importantes a tener en cuenta en seguridad. *[10]*

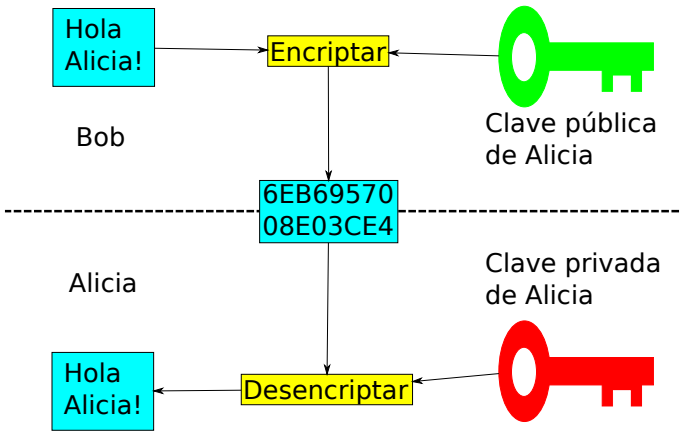

FIGURA 3.11: Esquema general del cifrado de clave pública

[7] Es, en esencia, el fundamento sobre el que se basa la firma digital.

3.10. RSA

Uno de los primeros algoritmos de criptografía pública (y el más utilizado hoy) es RSA (Rivest–Shamir–Adleman)[8]. Una de sus características más distintivas es el problema que plantea la factorización del producto de dos grandes números primos (*factoring problem*). El resultado de esta factorización es luego usado en la generación de las claves.

En general, los algoritmos de criptografía de clave pública son bastante más lentos que los de clave simétrica. Por ello, se suelen usar en conjunto con algún algoritmo de criptografía simétrica, como AES. *[26]*

3.10.1. Clave pública RSA

Dentro del conjunto del par de claves RSA, la pública es aquella que no mantenemos en secreto para que cualquiera que quisiera comunicarse con nosotros pudiera hacerlo de manera confidencial. Este tipo de claves consta de dos elementos:

- **n** – el módulo RSA, un entero positivo.
- **e** – el exponente público RSA, otro entero positivo.

Lo primero que se debe hacer para generar la clave pública es seleccionar de manera aleatoria dos números primos impares distintos, lo suficientemente grandes como para que sea computacionalmente inviable su descubrimiento por parte de un atacante. Una vez encontrados, el módulo n de la clave pública será el producto de estos dos números primos.[9]

$$n = p \cdot q$$

Ahora que tenemos el módulo de la clave definido, es hora de generar un exponente público e adecuado. Para ello, haremos uso de la función de Carmichael[10], la cual tiene la siguiente forma:

$$\lambda(n) = mcm(p-1, q-1)$$

[8] RSA debe su nombre a sus creadores: Ron Rivest, Adi Shamir y Leonard Adleman.

[9] Por convención, se denotan como p y q.

[10] La función que se muestra solo es válida cuando n es el producto de dos números primos impares y distintos.

El exponente público e será aquel número entero comprendido entre 3 y (n - 1) que satisfaga:

$$MCD(e, \lambda(n)) = 1$$

[11]

3.10.2. Clave privada RSA

La clave privada es la que mantenemos oculta. Aunque existen varios modelos, generalmente consta de dos elementos:

- **n** – el módulo RSA, un entero positivo (Debe ser el mismo que el de la clave pública).
- **d** – el exponente privado RSA, un entero positivo.

Para calcular el exponente privado d deberemos elegir un entero positivo menor que n, que además cumpla:

$$e \cdot d \equiv 1 \pmod{\lambda(n)}$$

, donde e será el correspondiente exponente público. *[12]*

3.10.3. Evaluación de la función RSA

Lo primero que debemos hacer es conseguir una representación de nuestro mensaje como un número entero, comprendido entre 0 y (n - 1).

Vamos a suponer que este mensaje queremos enviárselo a alguien cuya clave pública es (e, n). Si nuestro mensaje (recordemos que ahora es un entero) es M, para obtener el mensaje cifrado, aplicaremos la siguiente operación:

$$C \equiv E(M) \equiv M^e \pmod{n}$$

El entero C será del mismo tamaño que M y representará al mensaje cifrado.

Si ahora el destinatario quisiera descifrar el mensaje, solo tendría que revertir la operación con su clave privada (d, n) de la siguiente manera:

$$M \equiv D(C) \equiv C^d \pmod{n}$$

De esta manera, recuperaría el mensaje original.[11] *[15]*

3.11. RSASSA-PSS

Antes hablamos de la necesidad de preservar la integridad y la autenticación en las comunicaciones. RSASSA-PSS es un algoritmo de firma digital que sirve precisamente para ello. Combina RSA con un método de codificación llamado *Probabilistic Signature Scheme* (PSS).[12]

3.11.1. *Salt*

En criptografía, una *salt* es un fragmento de *bits* aleatorios que se utiliza junto a claves, funciones *hash* y otros mecanismos de cifrado con el fin de prevenir ataques de diccionario o de *rainbow table*. *[27]*

3.11.2. HMAC

Una HMAC (*Hash-based Message Authentication Code*) es una construcción usada para verificar la integridad de un mensaje, así como a su autor.

Hace uso de una función *hash* criptográfica, como MD5 o SHA-1, junto a una clave criptográfica secreta. La robustez de una HMAC depende de la función *hash* elegida, del tamaño del resumen y del tamaño de la clave. *[25]*

[11] Para cifrar con una clave privada y descifrar con una pública se hacen las mismas operaciones, cambiando e y d donde corresponda.

[12] Las siglas SSA corresponden a *Signature Scheme with Appendix*, lo que quiere decir que la firma va añadida al final del mensaje o junto a él.

3.11.3. *Probabilistic Signature Scheme* (PSS)

PSS es un método de codificación[13] desarrollado por Mihir Bellare y Phillip Rogaway, los cuales buscaban mejorar los métodos que existían. Para ello, incluyeron en su esquema el uso de una *salt*, lo cual haría más seguro el algoritmo frente a intentos de romperlo. *[2]*

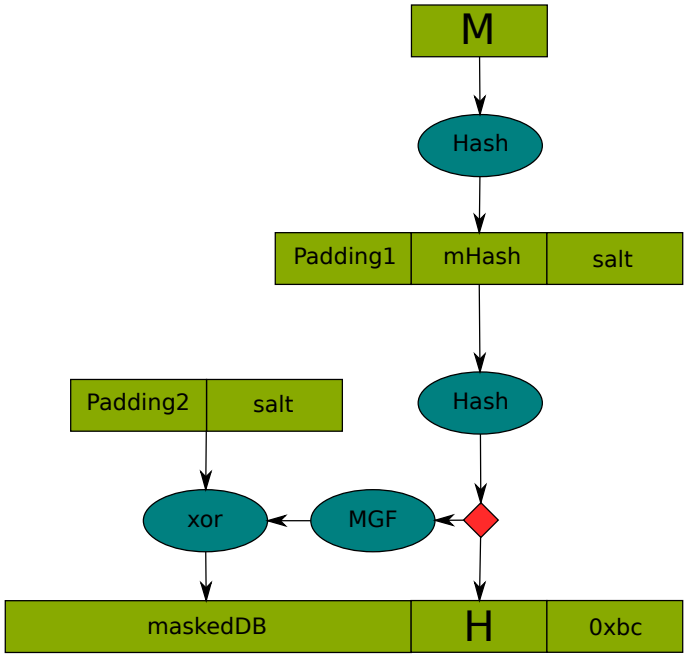

FIGURA 3.12: PSS de acuerdo a PKCS #1 V2.2 / RFC 8017

Como vemos en la Figura 3.12, PSS genera un resumen (*hash*) del mensaje. Este resumen se vuelve a pasar de nuevo por una función *hash*[14] junto a un *padding*[15] y la *salt* tal como muestra la figura.

El resultado de esta operación será la segunda de 3 piezas que conformarán nuestro mensaje codificado y la denotaremos como H. Para la primera (*maskedDB*), generaremos una máscara[16] de H y haremos una operación XOR con ella y la *salt* (junto a otro

[13]Un método de codificación es una operación que permite convertir un carácter de un determinado conjunto en un símbolo de otro sistema de representación.

[14]Las dos funciones *hash* que se usan en el esquema deben ser la misma.

[15]El primer *padding* estará formado por 8 *bytes* con valor *0x00*.

[16]Una máscara es muy parecida a una *hash*. Mientras la segunda tiene un tamaño determinado, una máscara puede tomar distintos tamaños según la necesidad.

padding).[17]

Ahora solo nos quedará añadir al final de nuestra salida un *byte* con valor *0xbc* y ya habremos acabado. *[13]*

3.12. HTTP

Hypertext Transfer Protocol (HTTP) es un protocolo de nivel de aplicación para sistemas de información distribuidos.

Es un protocolo genérico y sin estado que es usado para múltiples tareas como la transferencia de hipertexto o la representación de sistemas de ficheros.

Una característica de HTTP es la negociación de la representación de los datos, lo que permite construir sistemas independientemente de los datos que se vayan a transmitir. *[8]*

[17] Este *padding* está formado por una cantidad variable de *bytes* con valor *0x00*, teniendo al final un *byte* de valor *0x01*.

4 Diseño e implementación

4.1. Arquitectura de seguridad

Para entender la arquitectura de seguridad de la aplicación, es importante distinguir entre las siguientes propiedades de la seguridad de la información: confidencialidad, integridad y autenticación.

4.1.1. Confidencialidad

Para el cifrado de la información se utiliza criptografía de clave simétrica, ya que ofrece una mayor velocidad de cifrado frente a otros modelos. Debido a que puede funcionar tanto sobre *hardware* como sobre *software*, a que es un estándar del cifrado simétrico y a que no tiene debilidades conocidas, AES es el algoritmo de cifrado simétrico que se utiliza en esta aplicación. (Véase 3.8)

De entre todos los tamaños de clave disponibles se ha optado por el de 128 *bits*, ya que ofrece un nivel de seguridad adecuado para la aplicación. Además, un nivel superior habría supuesto una mayor carga computacional.

Igualmente, es necesario cifrar la clave simétrica utilizada. Para ello se utiliza un sistema de cifrado asimétrico para proteger la clave. Debido otra vez a que es un estándar, se utiliza RSA como algoritmo de cifrado asimétrico. (Véase 3.10)

Ya que el número de operaciones de cifrado asimétrico es menor que el del simétrico, se utiliza el tamaño máximo para una clave RSA: 4096 *bits*.

4.1.2. Integridad

También es importante que el mensaje permanezca íntegro. Para ello se incluye junto con el mensaje una HMAC. (Véase 3.11.2)

De entre algunos algoritmos de codificación se ha decidido utilizar PSS. Este algoritmo, entre otras cosas, combina el uso de una *salt* con un resumen *hash*, lo cual lo hace más robusto frente a determinados ataques enfocados en la obtención del mensaje original a partir de su resumen. (Véase 3.11.3)

4.1.3. Autenticación

El destinatario tiene que poder autenticar al autor del mensaje. Para lograrlo se incluye junto al mensaje una HMAC. (Véase 3.11.2)

Se utiliza como algoritmo de firma RSASSA-PSS, ya que combina los algoritmos utilizados para preservar la confidencialidad y la integridad. Este algoritmo genera un mensaje codificado a partir del resumen del mensaje original, utilizando para ello PSS, y luego firma (cifra) este mensaje usando RSA. (Véase 3.11)

Para proporcionar confidencialidad a la firma, esta va cifrada usando la clave pública del destinatario, de forma que nadie más pueda acceder a su contenido.

4.2. Arquitectura del software

El objetivo de este *software* es proporcionar una herramienta que permita a los usuarios mantener comunicaciones con otros usuarios preservando las tres propiedades de la seguridad comentadas en el punto anterior. Para ello, se ha llevado a cabo el desarrollo de varios prototipos:

- El primer prototipo sienta las bases de la arquitectura de orientación a objetos que utiliza la aplicación. Posee un esquema que recibe un fichero y lo divide en varios fragmentos de un tamaño dado, almacenando estos fragmentos en estructuras de datos. También puede recomponer el fichero original a partir de los fragmentos.
- El segundo prototipo realiza las mismas tareas que el anterior, y además añade confidencialidad a los fragmentos mediante un sistema de cifrado simétrico.

- El tercer prototipo realiza las mismas tareas que el anterior, y además añade el uso de RSA para cifrar claves simétricas y proporcionar integridad y autenticación a los fragmentos.
- El cuarto prototipo realiza las mismas tareas que el anterior, ahora como una aplicación en Android. Se añade una interfaz gráfica y hace uso de algunas herramientas que posee Android para el almacenamiento de claves.

4.2.1. Shatter I

El primer prototipo de la aplicación está desarrollado enteramente en Java, usando para ello el entorno de desarrollo Eclipse. Esta primera versión busca poder dividir un fichero en varios fragmentos de un tamaño dado, y luego poder recomponerlo. Para ello se han implementado algunas clases para almacenar los datos de los fragmentos:

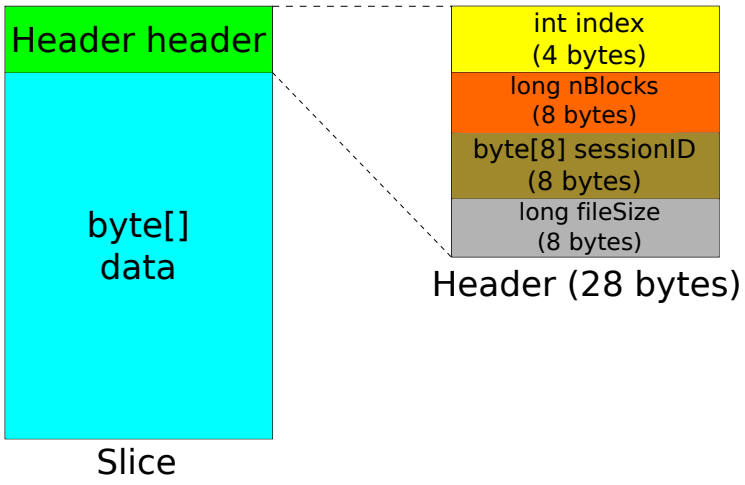

FIGURA 4.1: Esquema general de las clases `Slice` y `Header` (Versión 1)

- `Slice` – Un objeto de clase `Slice` es uno de los fragmentos en los que un fichero original se ha dividido. Está formado por una cabecera (`Header`) y un *array* de *bytes* en el que se almacena el contenido del segmento del fichero. (Figura 4.1) [1]

[1] Debido al contexto del TFG, se ha decidido no usar UML para confeccionar los esquemas expuestos en este capítulo.

- `Header` – En esta clase se almacenan los metadatos de un objeto de clase `Slice`. Se guardan datos como un contador, el número total de fragmentos para un fichero, un ID para la sesión[2] y el tamaño original del fichero. (Figura 4.1)

También se han desarrollado otras clases con métodos para trocear y recomponer un fichero:

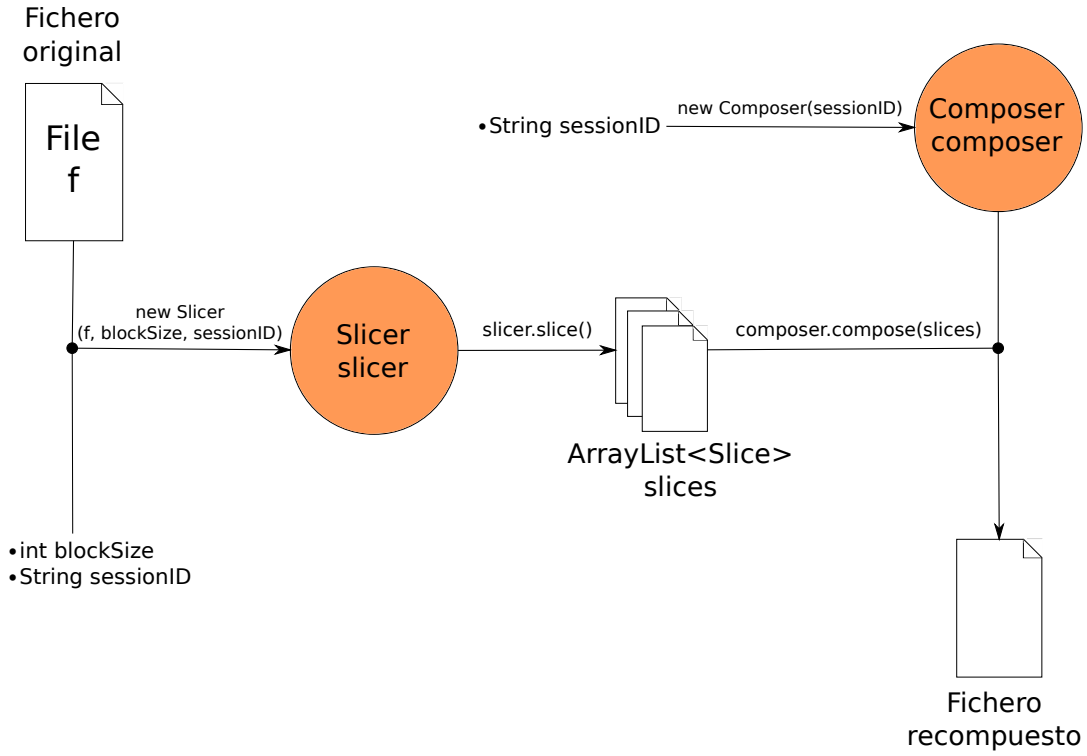

FIGURA 4.2: Esquema general del funcionamiento de las clases `Slicer` y `Composer`

- `Slicer` – Esta clase posee métodos para crear objetos de clase `Slice`. Recibe un fichero, un tamaño de bloque y un ID para identificar la sesión. Lee del fichero bloques del tamaño indicado hasta alcanzar el EOF y genera un `Slice` para cada uno de ellos, con una cabecera distinta. (Figura 4.2)
- `Composer` – Esta clase, a través de un método, recibe un *array* de objetos de clase `Slice` y devuelve un fichero compuesto. Lee uno a uno los fragmentos que recibe, prestando especial atención a sus cabeceras y si detecta que alguno falta genera un *log* de errores. (Figura 4.2)

[2]En esta primera iteración de la aplicación, el ID para identificar la sesión es un resumen *hash* del fichero.

4.2.2. Shatter II

En esta segunda versión, el objetivo del prototipo es proporcionar confidencialidad a los objetos de clase `Slice`. Para llevarlo a cabo se han implementado las siguientes clases:

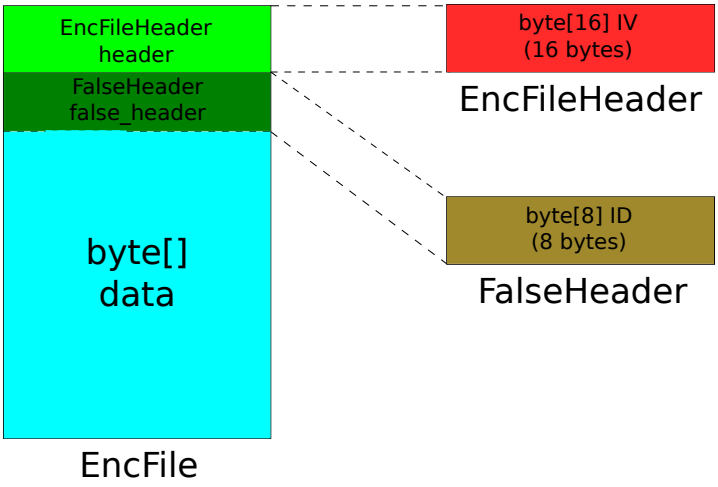

FIGURA 4.3: Esquema general de las clases `EncFile` y `EncFileHeader` (Versión 1)

- `EncFile` – Viene a ser un objeto de clase `Slice` cifrado. Aparte de los datos cifrados del `Slice`, también incluye una cabecera (`EncFileHeader`) con algunos datos importantes y una pseudocabecera (`FalseHeader`) con datos menores. (Figura 4.3)
- `EncFileHeader` – Esta cabecera se usa para almacenar algunos metadatos importantes como, en este caso, el vector de inicialización (IV) que se ha usado en el cifrado del objeto `Slice`. (Figura 4.3)
- `KeyFile` – Esta clase se utiliza para almacenar la clave simétrica que se ha utilizado para crear los objetos de clase `EncFile`.

Además de estas clases, se han creado otras clases que aportan los métodos necesarios para cifrar los fragmentos:

- `AESLibrary` – Básicamente, una clase que se encarga de generar de manera aleatoria y segura claves simétricas y vectores de inicialización.

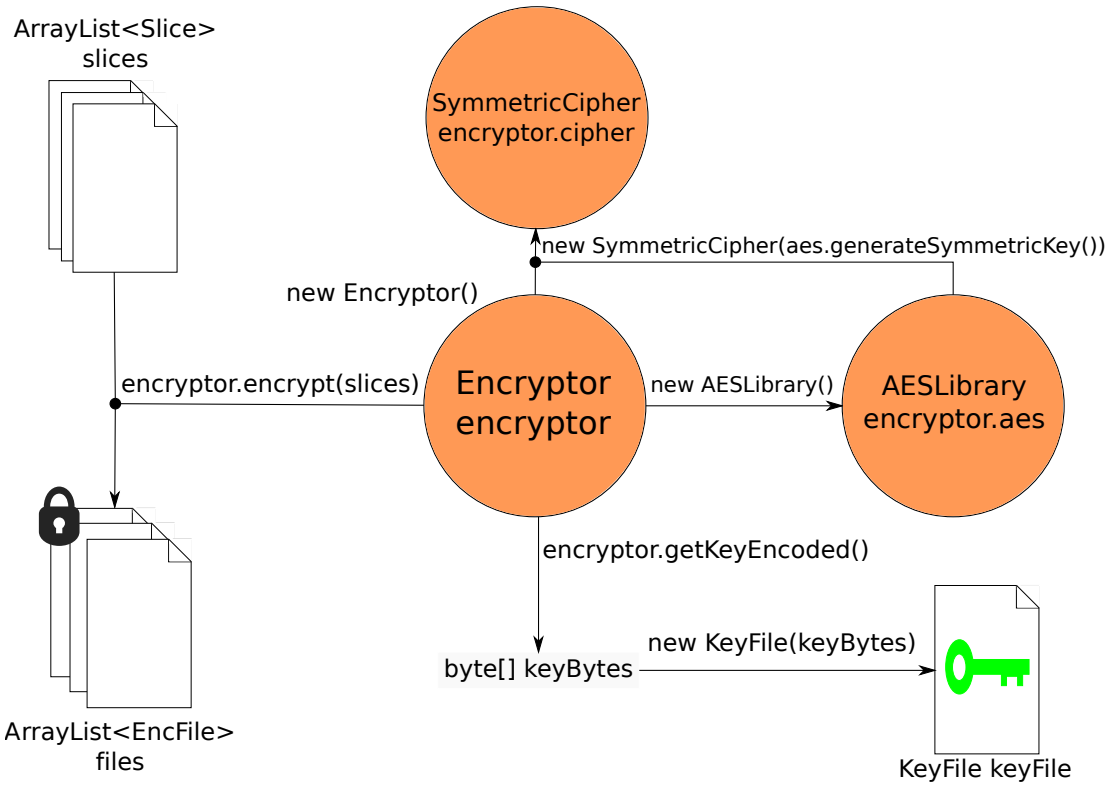

FIGURA 4.4: Esquema general del funcionamiento de la clase `Encryptor`

- `SymmetricCipher` – Esta es la clase que se encarga de hacer la parte más importante en cuanto a la confidencialidad. Una vez que ha sido inicializada con una clave simétrica, genera textos cifrados a partir de texto plano y un IV. Igualmente, puede llevar a cabo el proceso inverso.
- `Encryptor` – Esta clase, en conjunto con las dos anteriores, es la encargada de generar los objetos de clase `EncFile`. Genera de manera aleatoria y segura una clave simétrica para un algoritmo establecido y, con ella, inicializa una instancia de la clase `SymmetricCipher`. A continuación se le pasa un *array* de objetos de clase `Slice` del cual genera un *array* de objetos de clase `EncFile`, que es el que devuelve a través de un método. (Figura 4.4)
- `Decryptor` – El homólogo de la clase `Encryptor`. Realiza el proceso inverso y devuelve un *array* de objetos de clase `Slice` a partir de uno de objetos de clase `EncFile`. (Figura 4.5)

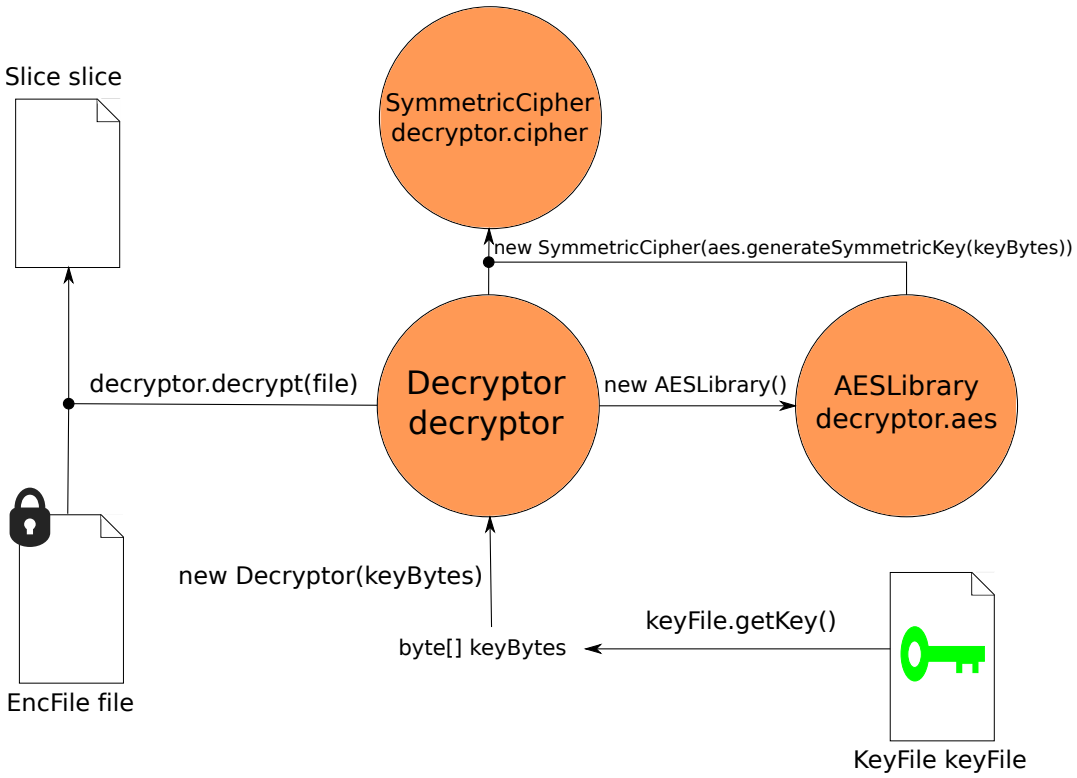

FIGURA 4.5: Esquema general del funcionamiento de la clase `Decryptor`

4.2.3. Shatter III

El objetivo en esta tercera iteración es el de proporcionar integridad y autenticación a las clases ya implementadas, al igual que confidencialidad a la clave simétrica que se utiliza para generarlas. Para lograrlo, se han implementado algunas clases nuevas:

- `EncKeyFile` – Esta clase almacena la clave simétrica protegida mediante cifrado asimétrico usando la clave pública del destinatario del mensaje, con lo que se logra la confidencialidad de la clave entre los usuarios. Tiene una cabecera (`EncKeyFileHeader`) en la que se guardan algunos datos importantes.
- `EncKeyFileHeader` – La cabecera de un objeto de clase `EncKeyFile`. En ella se almacena una HMAC cifrada de la clave simétrica.
- `Signature` – Básicamente una clase para contener una HMAC.
- `SecureSignature` – Contiene la HMAC comentada antes pero cifrada usando una clave asimétrica. De esta forma también le damos confidencialidad a la firma.

Para poder llevar a cabo el cifrado asimétrico se han desarrollado otras clases:

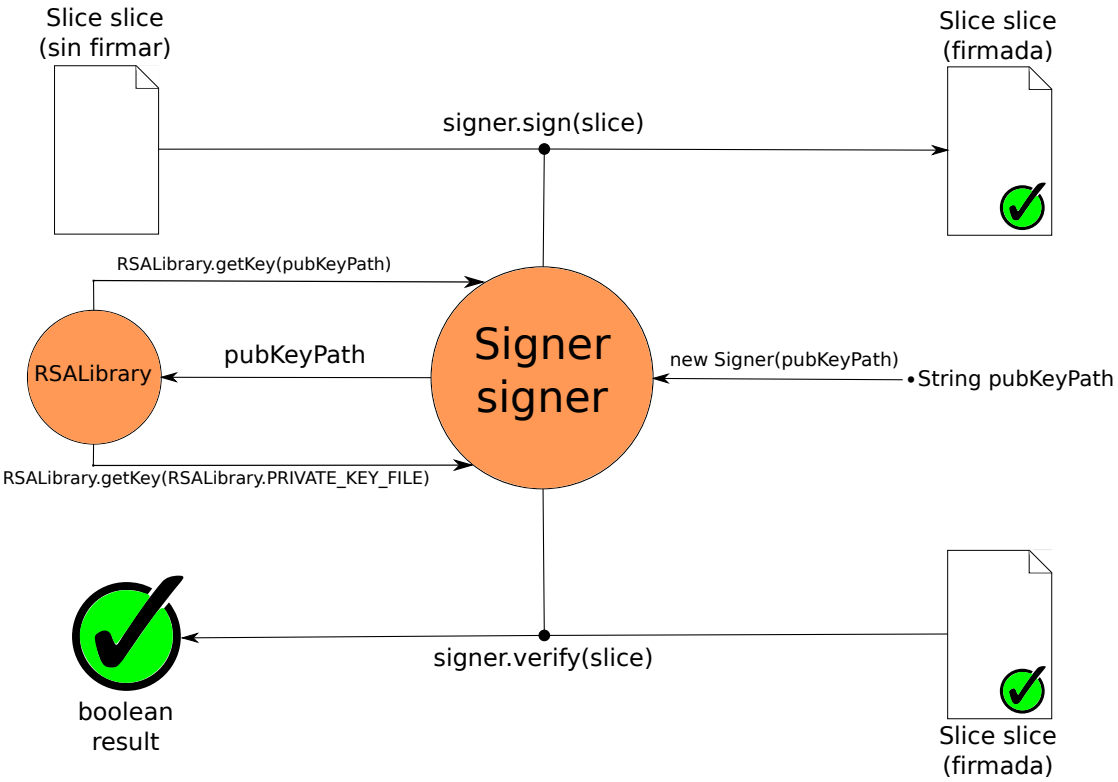

FIGURA 4.6: Esquema general del funcionamiento de la clase `Signer` (Versión 1)

- `RSALibrary` – Esta clase es la encargada de generar un par de claves asimétricas, de escribirlas y leerlas de un fichero, de cifrar un texto plano y de descifrar uno cifrado.
- `Signer` – La clase que se encarga de recibir objetos de clase `Slice`, `EncFile`, `KeyFile` y demás y devolverlos firmados. Asimismo, se encarga de comprobar las firmas de todas estas clases para preservar la autenticidad de la información que portan. (Figura 4.6)
- `RSAPSS` – Esta clase incorpora todos los métodos necesarios para generar, a partir de un texto plano, un texto codificado y firmado usando el algoritmo RSASSA-PSS. También realiza el proceso inverso y puede verificar las firmas.

En este prototipo se ha añadido una firma a varias de las clases que ya estaban implementadas:

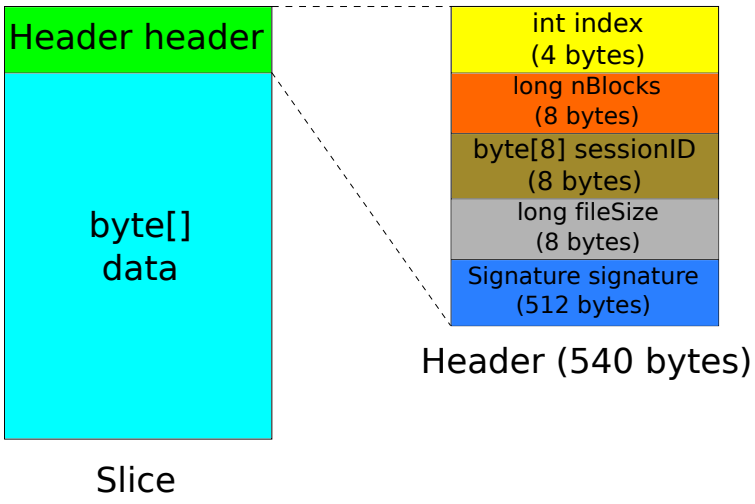

FIGURA 4.7: Esquema general de las clases `Slice` y `Header` (Versión final)

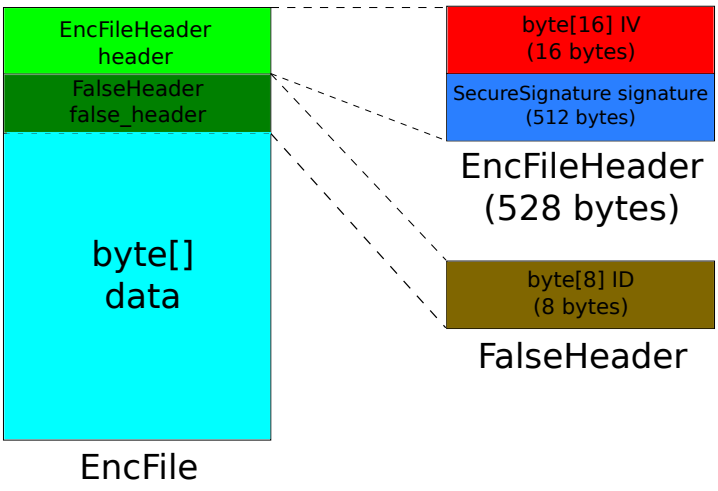

FIGURA 4.8: Esquema general de las clases `EncFile` y `EncFileHeader` (Versión final)

- A la clase `Slice` se le ha añadido en la cabecera una firma, que otorga integridad y autenticación a los datos que lleva. (Figura 4.7)
- A la clase `EncFile` se le ha añadido también una firma, esta vez para darle la integridad y la autenticación al IV que se encuentra en su cabecera. (Figura 4.8)
- La clase `KeyFile` tiene una firma de la clave simétrica.

Además de las clases mencionadas anteriormente, también se han desarrollado otras clases funcionales:

- `RandomString` – Esta clase posee un método que devuelve un `String` aleatorio de 8 caracteres.
- `FileIO` – Esta clase tiene métodos para escribir y leer los distintos ficheros que intervienen en la aplicación.
- `Bytes` – Tiene implementados varios métodos para hacer operaciones con *arrays* de *bytes*.

4.2.4. Shatter IV

Este último prototipo está desarrollado como una aplicación en Android. El objetivo de este prototipo es hacer uso de varias herramientas que posee Android para que la aplicación funcione correctamente. Para ello se han creado nuevas clases:

- `KeyStoreHandler` – Esta clase se usa para realizar todas las operaciones necesarias con el *Keystore* de Android (Véase 3.2.1). Se encarga de almacenar las claves, recuperarlas, usarlas para cifrar, firmar, etc.
- `HTTPClient` – Un cliente HTTP muy sencillo que únicamente realiza peticiones GET.
- `ExternalStorage` – Esta clase se encarga de interactuar con el *External Storage* de Android. Incorpora algunos métodos para conseguir *paths*, descriptores de fichero o crear directorios.

Para que los usuarios puedan enviar sus mensajes, se ha dispuesto un servidor HTTP bastante sencillo al que los usuarios pueden subir sus mensajes. Gracias a la clase `HTTPClient`, el destinatario puede bajarse los fragmentos de un mensaje.

El uso del *Keystore* de Android obliga a utilizar las claves que se almacenan en él usando unos métodos específicos. Debido a ello, algunas clases ya creadas (Como `RSALibrary` o `RSAPSS`) se sustituyen por métodos específicos que se implementan en la clase `KeyStoreHandler`. (Figura 4.9)

El ID que aparece en las cabeceras de la clase `Slice` se sustituye por un `String` aleatorio que genera la clase `RandomString`. Este cambio se debe a que el resumen *hash* que se usaba anteriormente queda en desuso al añadir la firma a estas clases. Además, este

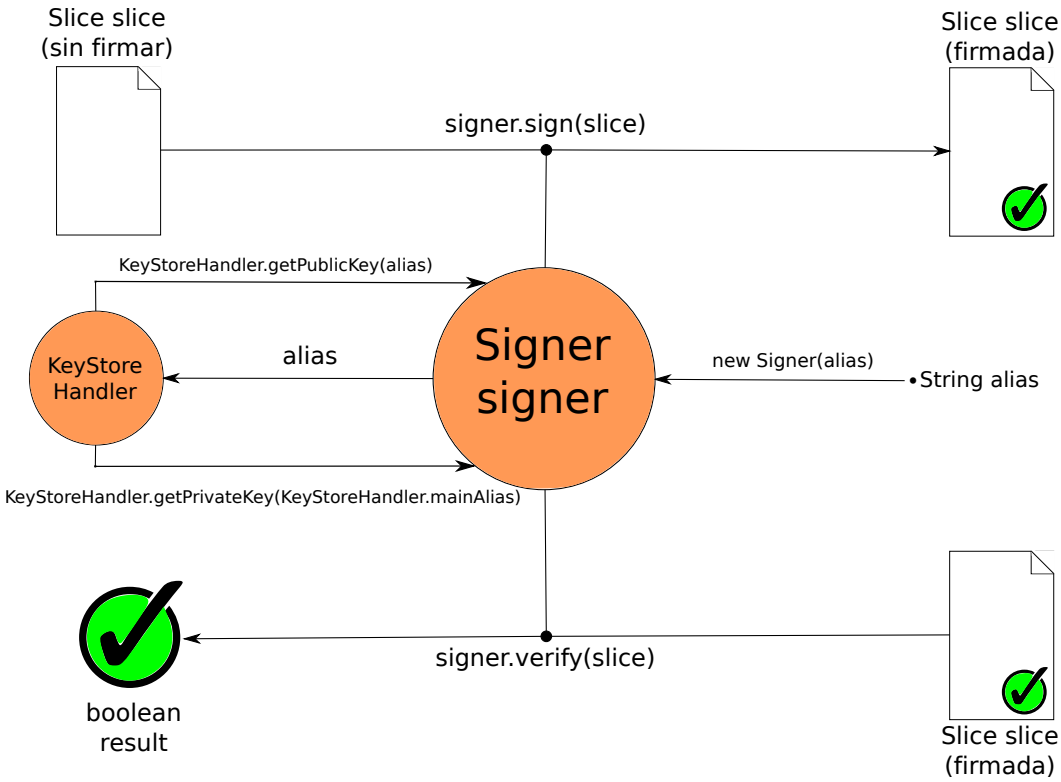

FIGURA 4.9: Esquema general del funcionamiento de la clase `Signer` (Versión final)

ID aleatorio identifica a todos los objetos de clase `Slice` que pertenecen a un mismo mensaje.

El esquema general de la aplicación se puede dividir en dos: Una primera parte se encarga de la división y cifrado del fichero (Figura 4.10). La otra se encarga de descargar del servidor los fragmentos cifrados y recomponer el fichero original (Figura 4.11).

4.3. Tiempo dedicado

El desarrollo de cada prototipo ha necesitado un tiempo de trabajo distinto. En esta sección se detalla el número de horas dedicado a cada uno de ellos:

- El primer prototipo, al tratarse de un esquema bastante sencillo, no ha supuesto un número de horas excesivo. El desarrollo de este prototipo se finalizó a las

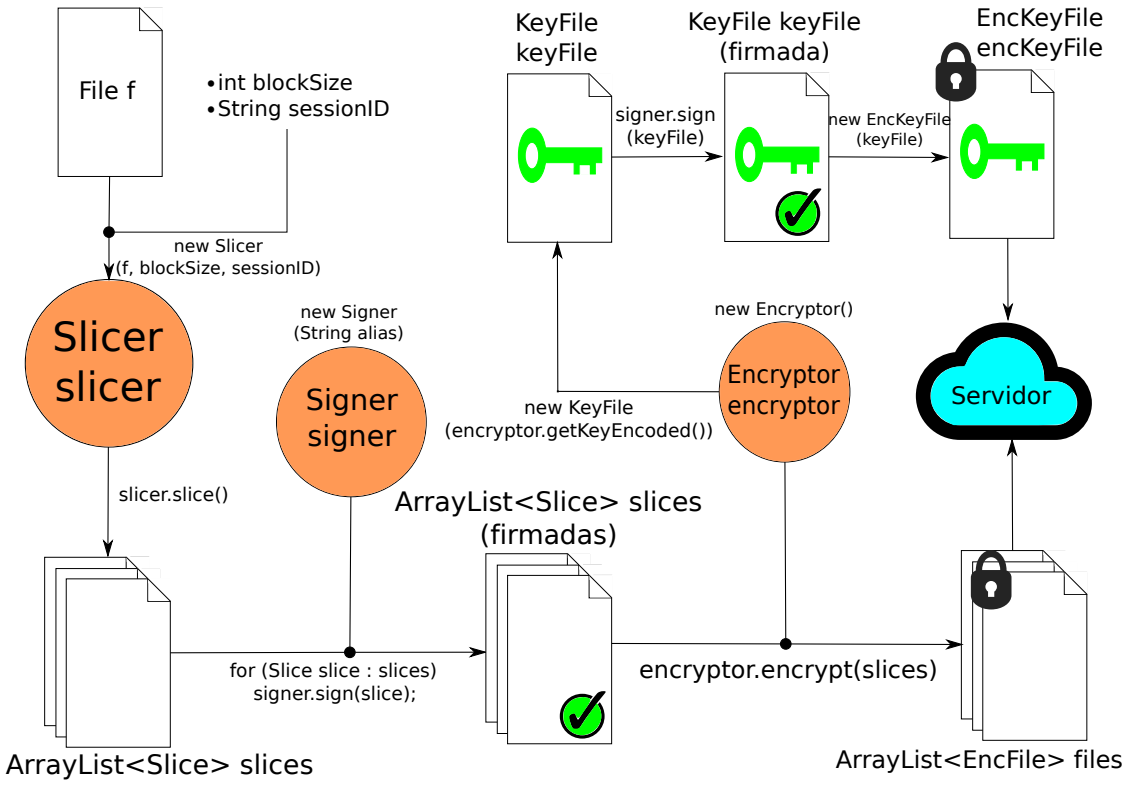

FIGURA 4.10: Esquema general de la clase `SliceEncrypt`, la primera parte en el proceso de comunicación de la aplicación

dos semanas de iniciar el proyecto, habiendo dedicado una media de dos horas diarias (28 horas).

- El segundo prototipo ha supuesto no solo tiempo de desarrollo, también ha necesitado tiempo de investigación para elegir el algoritmo más adecuado para el proyecto. Para este prototipo se dedicó, a partir de la finalización del prototipo anterior, dos semanas de desarrollo y una de investigación, dedicando una media de dos horas diarias (42 horas).

- El tercer prototipo ha necesitado una primera parte de investigación para elegir el algoritmo de cifrado asimétrico y firma adecuados y luego una parte de desarrollo. El desarrollo de este prototipo ha requerido, a partir de la finalización del prototipo anterior, dos semanas de investigación y cuatro de desarrollo, dedicando una media de dos horas diarias (84 horas).

- El cuarto prototipo, al tratarse de una migración a otra plataforma, no debería haber supuesto mucho tiempo de trabajo. Sin embargo se ha dedicado más tiempo

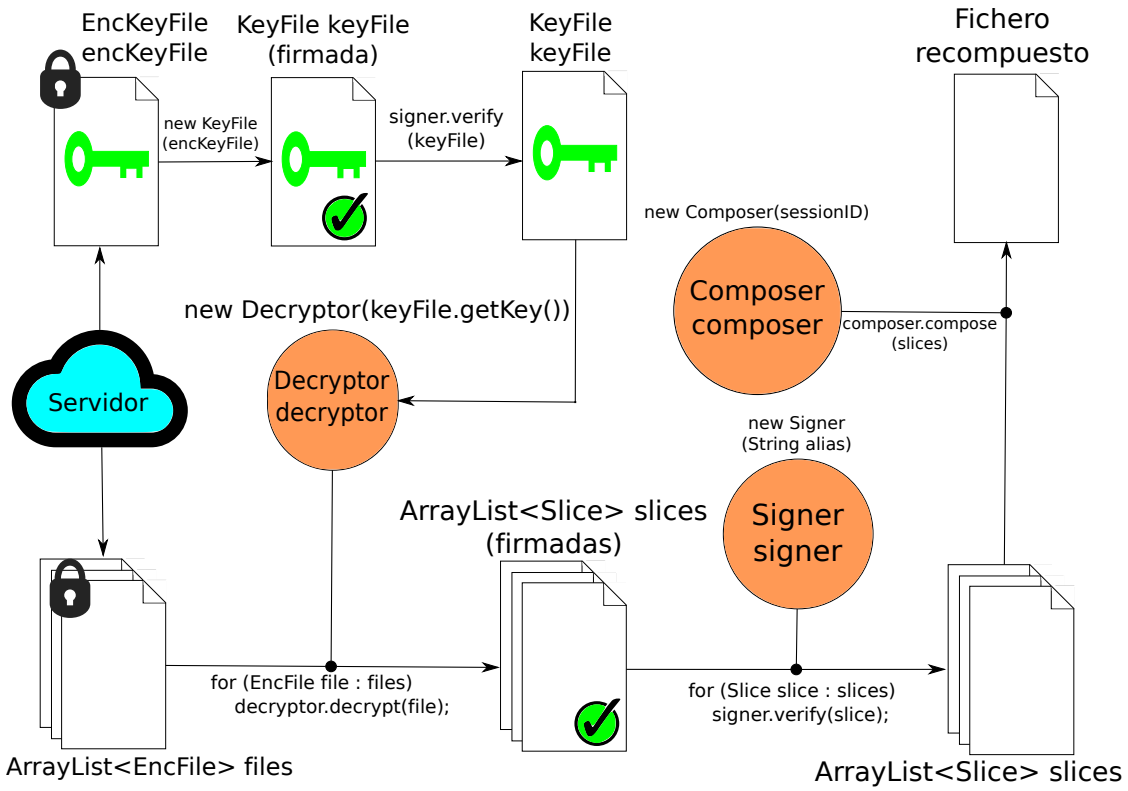

FIGURA 4.11: Esquema general de `DecryptCompose`, la parte final en el proceso de comunicación de la aplicación

debido a los problemas que se detallan en el Capítulo 5.2. Esta fase ha requerido una primera semana de investigación para encontrar la mejor forma de realizar la migración, y cinco semanas de desarrollo para llevarlo a cabo, dedicando una media de dos horas diarias (84 horas).

En total, el desarrollo del proyecto ha conllevado aproximadamente 238 horas de trabajo, habiendo invertido más tiempo en el desarrollo que en la investigación.

5 Resultados

5.1. Ejemplos de uso de la aplicación

De los prototipos mostrados en el Capítulo 4.2, el cuarto prototipo es la aplicación final. Esta sección muestra un ejemplo de uso completo de la aplicación real que se ha desarrollado, pasando por los prerrequisitos, el cifrado de un fichero y la descarga de un mensaje, todo acompañado de capturas de pantalla de la ejecución real.

FIGURA 5.1: Icono de la aplicación Shatter

5.1.1. Prerrequisitos

Al tratarse de una aplicación Android, es necesario el uso de un dispositivo que utilice el sistema operativo Android (Véase 3.2), específicamente la versión 6 (Marshmallow) o una superior.

También es necesario dar permisos de escritura en el almacenamiento del dispositivo, ya que la aplicación realiza operaciones de escritura y lectura.

5.1.2. Añadir contactos

Al abrir la aplicación por primera vez se genera un par de claves asimétricas, una pública y otra privada. En la pantalla principal, al no disponer de ningún contacto, solo se puede ver un campo de texto, un botón para seleccionar un fichero y un botón para añadir usuarios. (Figura 5.2)

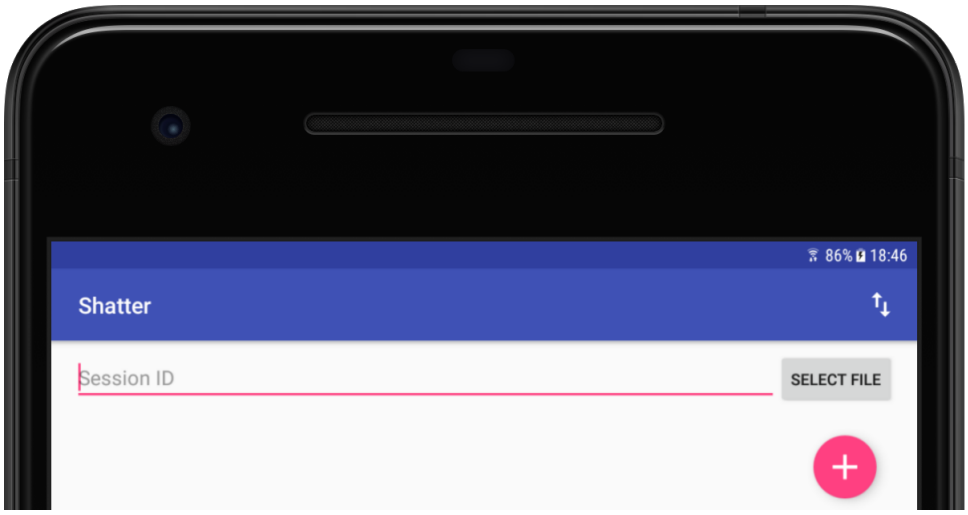

FIGURA 5.2: Pantalla principal de la aplicación

El primer objetivo es añadir algún usuario a la lista de contactos. Lo primero que hay que hacer es exportar la clave pública previamente generada haciendo uso del icono que se encuentra en la barra superior de la pantalla principal. (Figura 5.3)

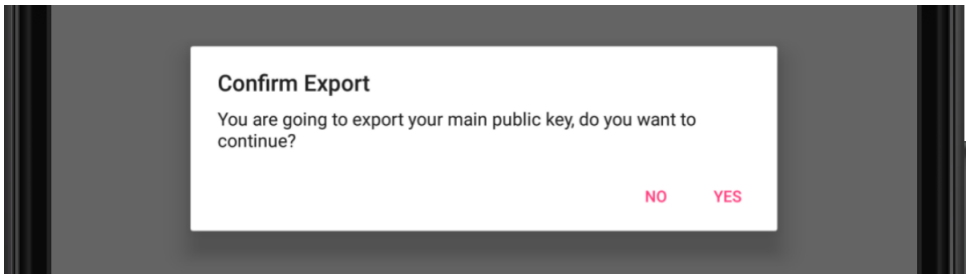

FIGURA 5.3: Mensaje de advertencia al exportar la clave pública

Al confirmar, se genera un certificado el cual contiene la clave pública antes generada en la ruta `Shatter/certs/main.crt`. [1]

[1] El directorio principal de la aplicación se encuentra en el almacenamiento externo del terminal y es accesible mediante un explorador de archivos.

Para añadir a un contacto se dispone de un botón en la esquina inferior derecha de la pantalla principal, el cual abre una nueva pantalla en la que se puede especificar un alias (nombre de usuario) y un certificado con la clave pública del nuevo usuario. (Figura 5.4)

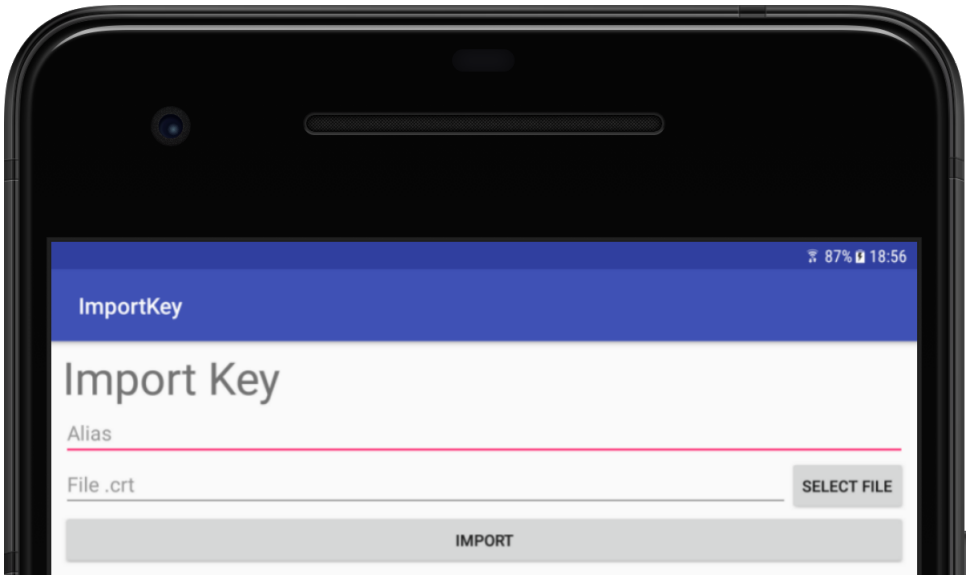

FIGURA 5.4: Pantalla para importar la clave pública de un nuevo usuario

Una vez el nuevo usuario se ha importado, la pantalla principal se actualiza y muestra el alias que tiene asignado, junto a unos botones con los que se puede operar con su clave pública. (Figura 5.5)

5.1.3. Cifrado y envío

Para enviar un mensaje a un contacto, se debe utilizar el botón *Select File*, el cual abre una nueva pantalla para elegir un fichero (Figura 5.6). El selector de ficheros pone en el campo de texto de la pantalla principal el *path* absoluto del fichero seleccionado (también se puede escribir a mano), y ya solo queda tocar el botón *Encrypt* del usuario al que vaya destinado el mensaje para que el fichero se fragmente y cifre. Si todo ha salido bien, se puede ver un mensaje en pantalla informando de ello, y en la ruta `Shatter/send/ID` se encuentran los fragmentos junto con la clave. (Figura 5.7)

El ID generado se debe comunicar al contacto al que vaya destinado, pero antes se debe subir el directorio que contiene los fragmentos al servidor haciendo uso del comando

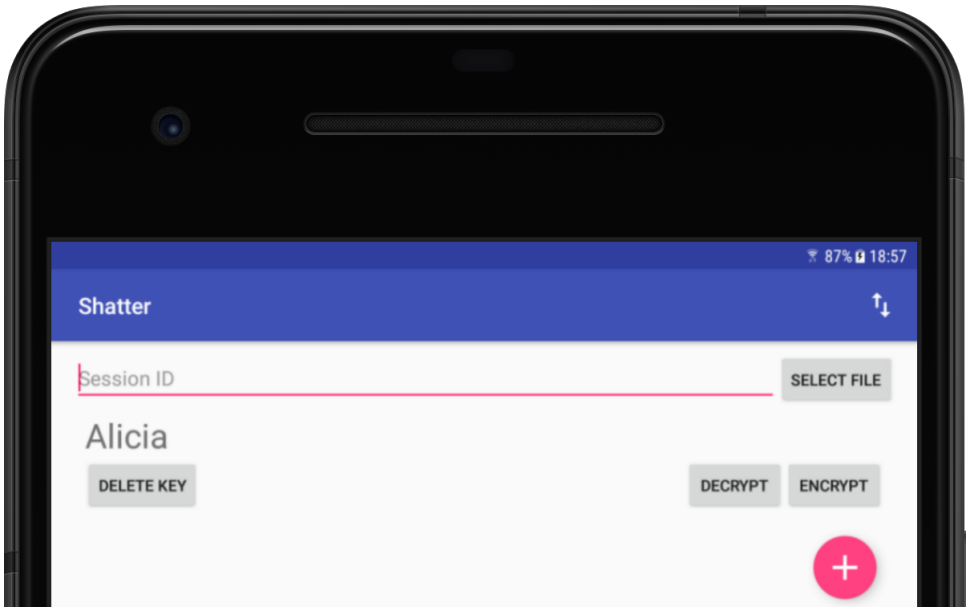

FIGURA 5.5: Pantalla principal de la aplicación con usuarios añadidos

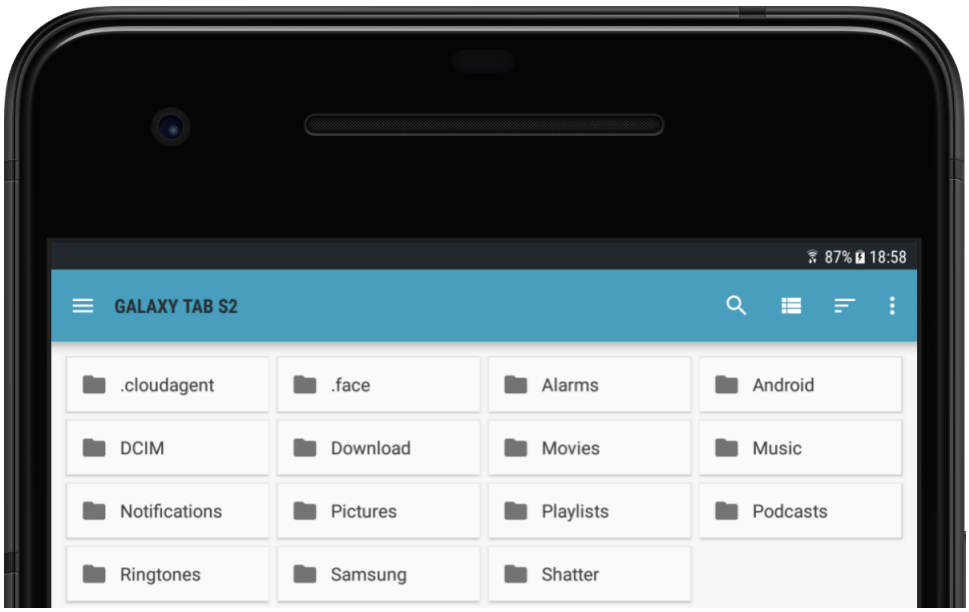

FIGURA 5.6: Pantalla para seleccionar un fichero

`scp` de Linux, por lo que se debe disponer de un usuario habilitado en la máquina que aloja el servidor. [2]

[2] En caso de estar cifrando el mismo mensaje para varios usuarios, se puede saber que ID corresponde a cada usuario mirando un registro que se encuentra en la ruta `Shatter/list.txt`

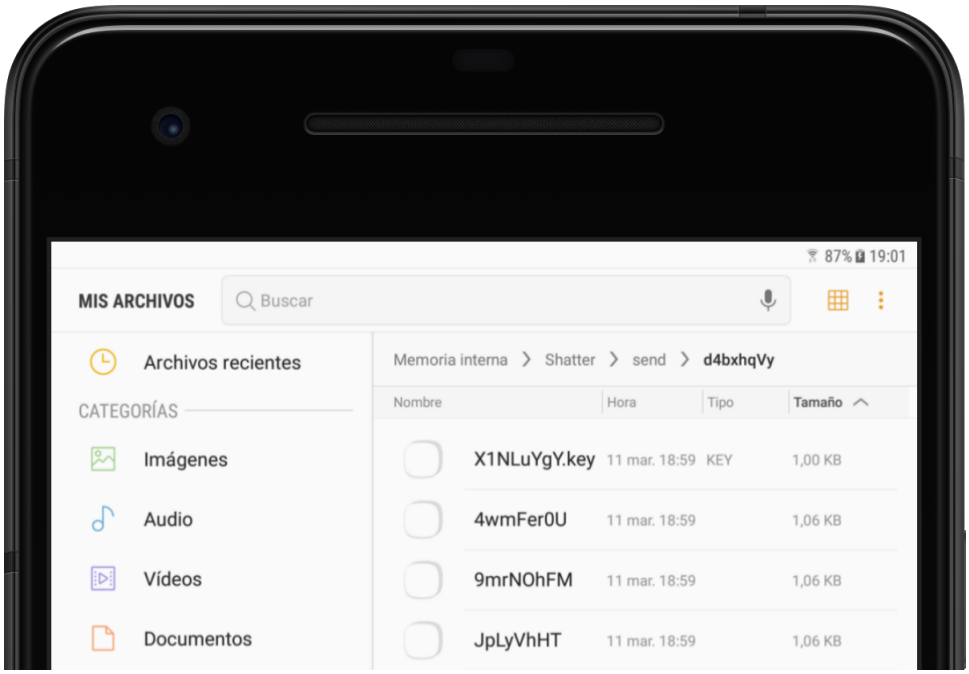

FIGURA 5.7: Fragmentos de un mensaje junto a su clave

```
$ scp Shatter/send/ID server
```

5.1.4. Descarga y descifrado

Para descargar un mensaje de un contacto determinado, este debe comunicar de alguna manera el ID del mensaje, el cual se debe escribir en el campo de texto de la pantalla principal. A continuación se pulsa el botón *Decrypt* asociado al alias del contacto.

Con esto, la aplicación pide al servidor un índice con todos los ficheros asociados a ese ID y descarga todos los que pueda en el directorio `Shatter/ID`. En caso de que algún fichero falte, informa de ello al usuario antes de proceder con el descifrado y rellena un registro indicando cuales son los fragmentos que faltan. (Figura 5.8)

Si todos los fragmentos y la clave son descargados exitosamente, la aplicación procede con el descifrado de los mismos. Durante el proceso se crean, en caso de ser necesario, ficheros de registro en los cuales se reflejan los problemas surgidos. En caso de no ocurrir ningún problema, el fichero original se recompone en la ruta `Shatter/ID/done/ID`. (Figura 5.9)

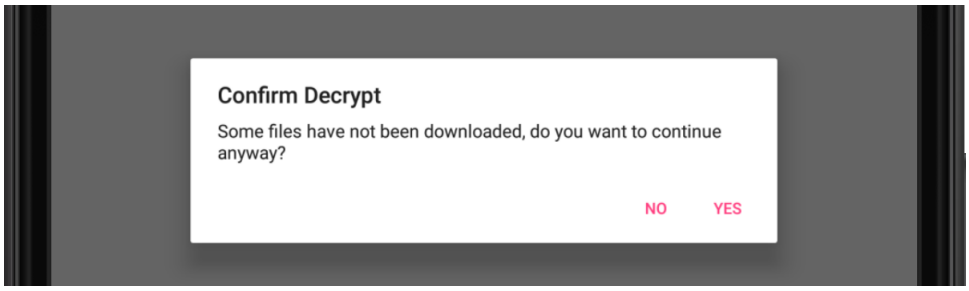

FIGURA 5.8: Mensaje advirtiendo de que algunos fragmentos no se han descargado

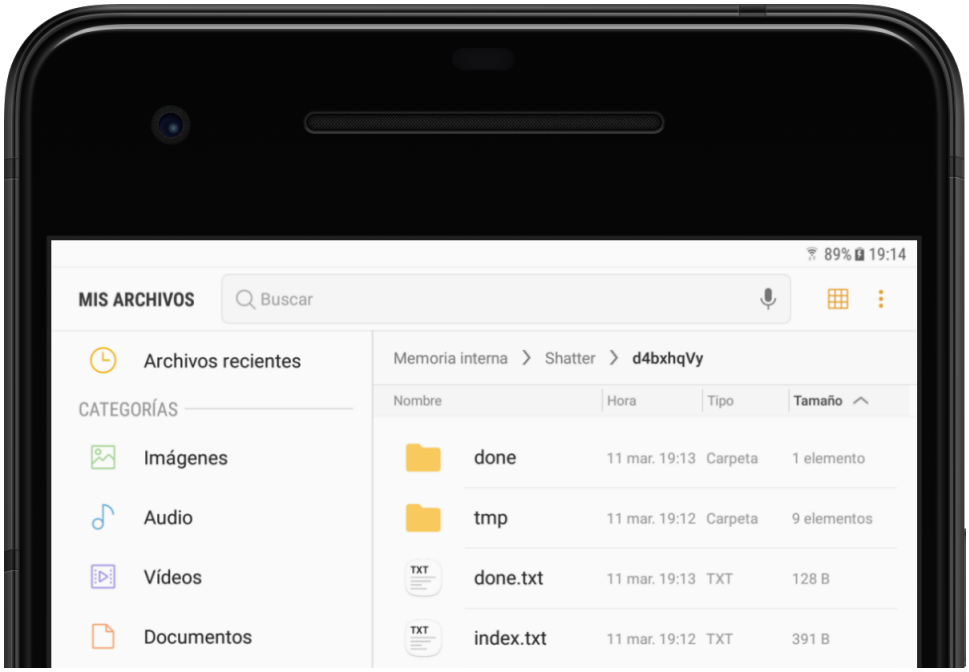

FIGURA 5.9: Directorio de un mensaje recibido

5.2. Problemas encontrados

A lo largo de la etapa que supuso el desarrollo de la aplicación, se han encontrado varios problemas que dificultaron la finalización del proyecto.

El mayor problema con el que se tuvo que lidiar es el no haber creado una parte portable que hiciera más fácil la migración a Android. Algunas librerías usadas cuando el proyecto solo era una aplicación escrita en Java tienen ciertas dependencias, las cuales

no se encuentran en ninguna librería Java usada en Android. Además, el uso del *Keystore* de Android para almacenar las claves exige realizar el cifrado asimétrico, las firmas y la generación de las claves de una manera determinada, lo que supuso la eliminación de las clases que se encargaban anteriormente de ello (`RSALibrary` y `RSAPSS`). Aunque algunos de estos problemas no se habrían podido solucionar aun teniendo una parte portable, sin duda habría supuesto un ahorro considerable de tiempo.

A raíz de lo anterior, la mayoría de las clases no partieron de un buen diseño. Esto supuso muchos cambios a lo largo del desarrollo de la aplicación (Cabeceras, modos de cifrado, registros...). Un mejor diseño original del programa habría ahorrado una gran cantidad de tiempo y energía.

Pero no todos los problemas encontrados tienen que ver con el diseño. La aplicación ahora cuenta con un servidor dedicado pero, en un principio, se pensó en utilizar un servicio externo para el almacenamiento de los mensajes enviados por los usuarios. De algunas ideas que salieron, se eligió la aplicación Pastebin[3], ya que permite la subida anónima de texto plano. La idea era que la aplicación subiera los distintos fragmentos cifrados a la plataforma, permitiendo al resto de usuarios su descarga. Sin embargo, la existencia de límites en el número de mensajes enviados o en la longitud de los mismos hicieron que se desechara la idea en favor de un servidor dedicado.

Debido a la inexperiencia en el campo que abarca este proyecto, es normal encontrarse con problemas de este tipo durante su desarrollo. Sin embargo, es importante señalar el valor didáctico de estos fallos, los cuales evitarán que se vuelvan a repetir en futuros proyectos.

[3] `https://pastebin.com/`

6 Conclusiones

6.1. Consecución de objetivos

Los objetivos establecidos al comienzo del proyecto son:

- Proporcionar un mecanismo que permita la transmisión de información entre distintos dispositivos.
- Desarrollar un esquema que permita cifrar y descifrar la información que se quiere transmitir.
- Diseñar una forma de comprobar la integridad y la autenticación de la información.

Estos objetivos se han cumplido de manera satisfactoria: se ha creado una vía para que distintos usuarios puedan intercambiar información, estando esta convenientemente cifrada y firmada.

Sin embargo, el objetivo principal del proyecto es crear una aplicación accesible a cualquier usuario acostumbrado a un sistema de mensajería. Este objetivo no se ha cumplido: la aplicación resultante requiere tener un usuario habilitado en el servidor para enviar los fragmentos mediante `scp`, y el intercambio de claves resulta demasiado engorroso si lo comparamos con otras aplicaciones parecidas, ya que requiere acceder al sistema de ficheros del dispositivo para extraer los certificados.

6.2. Conocimientos adquiridos

Desde el comienzo del proyecto hasta la finalización de esta memoria he estado inmerso en un continuo aprendizaje. Aunque la mayoría de las competencias aprendidas

provienen de los conceptos clave que se abordan en este trabajo, otros han sido adquiridos mediante ensayo y error en el desarrollo.

Los conocimientos más relevantes que he adquirido son:

- Reforzar y ampliar la mayoría de los conceptos e ideas introducidos en la asignatura de seguridad impartida en el grado.
- Poder estudiar el problema que supone la factorización de números primos en la generación de claves RSA.
- Comprender mejor los estándares AES y RSA estudiando sus RFC correspondientes y pudiendo ver como funcionan internamente.
- Aprender como funciona internamente el algoritmo de firma RSASSA-PSS, llegando a desarrollar el algoritmo paso a paso, siguiendo el RFC.
- Ampliar la experiencia en Android al desarrollar la aplicación y todas las pantallas que incluye.
- Abordar un proyecto grande de desarrollo, teniendo que pasar por varias etapas para la organización, la investigación y, finalmente, el desarrollo.
- Ampliar las habilidades sobre LaTeX, al utilizarlo en la confección de esta memoria.

Todo esto me ha permitido comprender mejor el mundo del desarrollo en la seguridad informática, al menos la parte que corresponde a la criptografía. Aunque esperaba que el proyecto tendiese más hacia la investigación, no estoy para nada disgustado, y me quedo con haber adquirido una nueva perspectiva de este campo.

6.3. Líneas de desarrollo futuras

Las líneas de desarrollo futuras se centran en mejorar la accesibilidad y la interfaz de la aplicación.

Estas nuevas iteraciones se basan en añadir más funcionalidad al intercambio de claves, como la generación de un código QR que permite a un usuario dar a conocer su clave pública.

Algunos ajustes de personalización como poder modificar el tamaño de bloque de cifrado, una interfaz más vistosa o un sistema de notificaciones son otras mejoras que añadir en siguientes versiones.

Por último, un cambio que hace más sencillo el uso de la aplicación es modificar la manera en la que se suben los fragmentos al servidor (actualmente se hace mediante `scp`) por un sistema que permite hacer llamadas POST.

Todo el código utilizado en este TFG está disponible en un repositorio de GitHub[1].

[1] `https://github.com/merinhunter/Shatter`

Índice alfabético

Bibliografía

[1] Martín Balao. *Criptografía: Padding + ECB + CBC*. 2011. URL: http://martin.com.uy/sec/criptografia-padding-ecb-cbc/.

[2] Mihir Bellare y Phillip Rogaway. *PSS: Provably Secure Encoding Method for Digital Signatures*. 1998. URL: http://grouper.ieee.org/groups/1363/P1363a/contributions/pss-submission.pdf.

[3] The Legion of the Bouncy Castle. *BC Home*. 2013. URL: http://bouncycastle.org/.

[4] Thomas W. Cusick y Pantelimon Stanica. «Cryptographic Boolean functions and applications». En: ed. por Academic Press. 2009. Cap. 7, págs. 158-159.

[5] Hans Delfs y Helmut Knebl. *Introduction to cryptography: principles and applications*. Ed. por Ueli Maurer. Springer, 2007.

[6] Android Developers. *Android Keystore System*. 2018. URL: https://developer.android.com/training/articles/keystore.html.

[7] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation*. Inf. téc. 800-38A. NIST, 2001. URL: http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf.

[8] R. Fielding y col. *Hypertext Transfer Protocol – HTTP/1.1*. 1999. URL: https://tools.ietf.org/html/rfc2616.

[9] James Gosling y col. *The Java® Language Specification*. 2015. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf.

[10] Frederick J. Hirsch. *SSL/TLS Strong Encryption: An Introduction*. 2013. URL: http://httpd.apache.org/docs/2.2/ssl/ssl_intro.html#cryptographictech.

[11] Ed. K. Moriarty y col. *PKCS #1: RSA Cryptography Specifications Version 2.2*. Inf. téc. RFC 8017. IETF, 2016, págs. 8-9. URL: https://tools.ietf.org/html/rfc8017#section-3.1.

[12] Ed. K. Moriarty y col. *PKCS #1: RSA Cryptography Specifications Version 2.2*. Inf. téc. RFC 8017. IETF, 2016, págs. 9-10. URL: https://tools.ietf.org/html/rfc8017#section-3.2.

[13] Ed. K. Moriarty y col. *PKCS #1: RSA Cryptography Specifications Version 2.2*. Inf. téc. RFC 8017. IETF, 2016, págs. 42-43. URL: https://tools.ietf.org/html/rfc8017#section-9.1.1.

[14] NIST. *Announcing the ADVANCED ENCRYPTION STANDARD (AES)*. Inf. téc. FIPS 197. NIST, nov. de 2001. URL: http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[15] R. L. Rivest, A. Shamir y L. Adleman. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. 1978. URL: http://people.csail.mit.edu/rivest/Rsapaper.pdf.

[16] Wikimedia. *File:AES-AddRoundKey.svg*. 2006. URL: https://commons.wikimedia.org/wiki/File:AES-AddRoundKey.svg.

[17] Wikimedia. *File:AES-MixColumns.svg*. 2006. URL: https://commons.wikimedia.org/wiki/File:AES-MixColumns.svg.

[18] Wikimedia. *File:AES-ShiftRows.svg*. 2006. URL: https://commons.wikimedia.org/wiki/File:AES-ShiftRows.svg.

[19] Wikimedia. *File:AES-SubBytes.svg*. 2006. URL: https://commons.wikimedia.org/wiki/File:AES-SubBytes.svg.

[20] Wikimedia. *File:Android robot 2014.svg*. 2014. URL: https://commons.wikimedia.org/wiki/File:Android_robot_2014.svg.

[21] Wikimedia. *File:Java programming language logo.svg*. 2017. URL: https://en.wikipedia.org/wiki/File:Java_programming_language_logo.svg.

[22] Wikipedia. *Advanced Encryption Standard*. 2017. URL: https://en.wikipedia.org/w/index.php?title=Advanced_Encryption_Standard&oldid=809193008.

[23] Wikipedia. *Android (operating system)*. 2017. URL: https://en.wikipedia.org/w/index.php?title=Android_(operating_system)&oldid=805639596.

[24] Wikipedia. *Block cipher mode of operation*. 2017. URL: https://en.wikipedia.org/w/index.php?title=Block_cipher_mode_of_operation&oldid=809021817#Common_modes.

[25] Wikipedia. *HMAC*. 2018. URL: https://en.wikipedia.org/w/index.php?title=HMAC&oldid=832688082.

[26] Wikipedia. *RSA (cryptosystem)*. 2017. URL: https://en.wikipedia.org/w/index.php?title=RSA_(cryptosystem)&oldid=806096352.

[27] Wikipedia. *Salt (cryptography)*. 2018. URL: https://en.wikipedia.org/w/index.php?title=Salt_(cryptography)&oldid=828398021.

[28] Wikipedia. *Threat model*. 2017. URL: https://en.wikipedia.org/w/index.php?title=Threat_model&oldid=810785371.